By signing my BannerID below, I affirm and certify that (1) I received no information from students who took the following exam early, and (2) I will give no information to students who are taking the exam at a later date.

Banner ID (written legibly):

Please do NOT put your name or login on this exam. Only Banner ID

This is the first exam for CS16. It starts at 7 PM and ends at 12 midnight. This exam is meant to take about 2-3 hours, but you may stay for up to 5 hours. We strongly recommend leaving by 11 PM; chances are that anything you add after that time will reduce rather than increase your grade.

- You may not use any resources except your mind and your writing implement (unless you have pre-discussed accommodations): no books, phones, computers, etc.

- You may ask the course staff for clarifications of problems.

Please do the following:

1. Read through the entire exam carefully, and make sure you understand what each question is asking. Please note the general guidelines on the next page.

2. You have plenty of time, so please write your answers clearly and neatly on the exam paper. For all problems, but especially for pseudocode, you may want to write a first and maybe even a second draft on a piece of scrap paper, and then write a clean copy on the exam.

3. We will only grade the work in this exam booklet or on scratch paper stapled to the exam.

4. Check your work carefully.

Good luck!

<table>
<thead>
<tr>
<th>Problem</th>
<th>Possible Points</th>
<th>Score</th>
<th>TA Login</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>100</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
General assumptions and guidelines:

1. For pseudocode, you may assume that your inputs are valid (e.g. if we say that your function takes in an array of increasing numbers, you do not need to check that the array is not null or that the items are numbers or that they are increasing).

2. For pseudocode, you are free to write helper functions as needed. However, do not alter the input parameters that are stated for the functions given in the problem.

3. Unless specifically told otherwise, you may assume that you have access to general-purpose data structures and their methods that we have talked about in class. For example, if you would like to use a priority queue with $O(\log n)$ insert and removal of minimum element, you needn’t reimplement heap. You may simply create an instance of the data structure.

4. We will be grading based on correctness, clarity/succinctness, and efficiency. You could turn in perfectly-functional pseudocode that does not receive full credit because it is unreasonably cryptic or not the most efficient way to solve the problem.
The real test will consist of problems similar to some of the problems listed below and
draw from the following topics. A really good way to study is to do all the problems
below independently and then compare your solutions with a friend. If you actually solve
the problems with friends, be sure you understand the solutions, and how to get to the
solutions yourself. Memorizing the solutions to these problems will be of almost no value
on the actual test. Make sure none of the following topics are foreign to you.

**Topics to Study**

Anything covered in lecture, projects, or homeworks before starting the graphs unit is
fair game! Here is a list of topics you should be familiar with!

- Dynamic Programming
- Analysis and Big-O, Big-Omega, Big-Theta
- Amortized Analysis, Expected Analysis
- Expanding Data Structures (Stacks, Queues)
- Hashing, Sets, and Dictionaries
- Arrays, Binary Search
- Trees and Traversals
- Binary Search Trees
- Priority Queues, Heaps
- Sorting, Master Theorem
- Selection

**Topics You Are Not Responsible For**

- Hashing proof
- Proving expected runtimes (you should know what the expected runtimes of various
  algorithms/data structures are, but you will not need to prove that something has
  a certain expected runtime)

- Graphs
- DAGs and Topsort
- Decision Trees
1 Recurrence and Induction

The following code messes with the entries of an array, and then recursively calls itself on a smaller array. You can assume that the input array size \( n \) is a power of two.

```python
def transform(arr):
    """transform: int array \rightarrow\ int array
    Purpose: takes an array of \( n \) ints, where \( n \) is a power of 2, and returns an array
    ""
    if len(arr) == 1:
        return arr
    half = len(arr)/2
    sums = [0] * half  # build arrays of n/2 zeroes
    diffs = [0] * half
    for i in range(0, half):
        a = arr[2 * i]
        b = arr[2 * i + 1]
        sums[i] = a + b
        diffs[i] = a - b
    return transform(diffs) + sums  # Note: list1 + list2 concatenates
                                # the lists: [1]+[2,3] \Rightarrow [1,2,3]
```

(a) Let \( T(n) \) be the running time of transform on any array of size \( n \). Write a recurrence relation for \( T \). Be sure to include a base case.

(b) Use plug-n-chug with \( n = 1, 2, 4, \) and 8 to conjecture a big-O solution to the recurrence.

(c) Here’s a recurrence relation for a different function, \( S \):

\begin{itemize}
  \item \( S(1) = 1 \)
  \item \( S(n) = n^2 + S\left(\frac{n}{2}\right) \)
\end{itemize}

The solution to this recurrence (for \( n \) a power of 2) is:

\[
S(n) = \frac{4n^2 - 1}{3}
\]

Prove this by induction; you need only handle the case where \( n \) is a power of two, i.e., where \( n = 2^k \) for some non-negative integer \( k \).

(d) Is the function \( S \) \( O(1) \)? Is it \( O(n^3) \)? Is it \( \Omega(n^2/2) \)? Is it \( \Theta(n) \)? Explain each case.
2 Tree Induction

(a) A (0,2) binary tree $T$ is one in which every node has out-degree zero or two, i.e., it has either two children or it’s a leaf. Give a recursive definition of a (0,2) binary tree. Take a look at HW5 for an example of a recursive definition of a tree.

(b) Prove by induction that the number of nodes in a regular binary tree is one more than the number of edges.
3 Amortized Analysis

As you saw in Homework 1, you can implement a queue with very little additional work by using two stacks, in and out. They both start out empty. To enqueue an item, you push it onto in. To dequeue an item, you pop it from out. Of course, that depends on out containing something! If out is empty, you first “pour” all the items from in into out, and then pop from out. This is what “pouring” looks like:

def pour():
    while not in.empty():
        out.push(in.pop())

(a) Draw a picture to indicate the state of the stacks in and out in an empty queue to which the following operations are applied: enq(A), enq(B), deq(), enq(C), deq(), deq(). You should draw a total of seven pictures, the first and last showing two empty stacks.

(b) Explain why enqueuing is worst-case $O(1)$ and dequeuing is worst-case $O(n)$, where $n$ is the number of items in the queue.

(c) Explain why the amortized cost of dequeuing, in any sequence of $n$ operations on an empty queue, is $O(1)$.
4 Hashing

What is a good hash function? What’s a bad one? What are hashtables and how do they work? Explain the difference between a hashset and a hashtable.
5 Search

You’re given a function $f$ that takes positive integers and returns integers in constant time. It has a special form, however: for the first “few” positive integers, the returned values are all positive, and in non-decreasing order (not necessarily increasing). But after some integer $n$, the function always returns $-1$. For instance, the code might look like this:

```python
def f(u):
    if (u < 4):
        return u*u + u + 3
    else
        return -1
```

In this case, $n$ is 3: for every $u > 3$, we have $f(u) = -1$, and for $1 \leq u \leq 3$, $f(u)$ is positive and non-decreasing (in fact, it’s increasing).

Given a positive integer $p$ and the function $f$, you must determine whether there is a positive integer $b$ where $f(b) = p$.

You may call the function $f$ as often as you need to.

Write pseudocode for an algorithm that solves the problem in $O(\log n)$ time. Remember that you are given the ability to invoke the function $f$, but you have no idea, a priori, what the value of $n$ might be. Note also that $f$ is defined for all positive integers, so the domain of $f$ is infinite.

**Hint:** It may be helpful to approach this problem in two parts. First, you want to search for the integer $n$ after which the function returns $-1$. Because $f(b)$ returns a positive integer, you know that $b$, if it exists, must lie within the first part of your domain (where $1 \leq u \leq n$).

Then, you want to search the first part of your domain for the integer $b$ for which $f(b) = p$. This process should sound familiar...